EUI WORKING PAPERS IN POLITICAL AND SOCIAL SCIENCES

EUI Working Paper SPS No. 92/19

RZoo: Efficient Storage and Retrieval of Social Science Data

GÖTZ ROHWER

European University Institute, Florence

Please note
As from January 1990 the EUI Working Paper Series is divided into six sub-series, each sub-series is numbered individually (e.g. EUI Working Paper **LAW** No 90/1).

EUROPEAN UNIVERSITY INSTITUTE, FLORENCE

DEPARTMENT OF POLITICAL AND SOCIAL SCIENCES

EUI Working Paper **SPS** No. 92/19

RZoo: Efficient Storage and Retrieval of Social Science Data

GÖTZ ROHWER

BADIA FIESOLANA, SAN DOMENICO (FI)

© Götz Rohwer
Printed in Italy in June 1992
European University Institute
Badia Fiesolana
I-50016 San Domenico (FI)
Italy

This text gives a description of RZOO, a program for retrieving numerical data from compressed ZOO archives. After a short Introduction, Section 2 describes how to setup an archive that can be used with RZOO. Section 3 describes the commands to control RZOO. Section 4 gives detailed information about the operation of the program. Finally, Section 5 discusses some technical details, memory requirements of the program and some aspects of its portability. The explanations are illustrated with a small example data set available on the program diskette.

The program was developed as part of the research project *Household Dynamics and Social Inequality* supported by grants from the *European University Institute (Florence)* and the *European Commission (Brussels)*. It is used in this project to organize, and to give simple and efficient access to the data of the German Socio-economic Panel. I wish to thank Hans-Peter Blossfeld for the opportunity to do the work in the context of this research project.

Contents

1 Introduction

In empirical research very often large sets of data must be handled. For instance, the data of the first six waves of the *German Socio-economic Panel* consist of about 50 files with, altogether, more than 105 megabytes. Putting such data in a compressed archive can free up considerable portions of storage space (and in addition, access times will be shorter). For example, the just mentioned panel data can be compressed to less than 12 megabytes.

However, compressed archives, created with standard software packages, have a major drawback. Whenever some part of the data is needed, all data files involved have to be decompressed before they can be used. Clearly, it would be very nice if we could have direct access to the compressed data without the necessity of prior decompression. Exactly this is done by RZOO. The program was developed to give direct access to the (numerical) data of a compressed archive of data files.

Generating compressed archives of data files can be done with many different archive programs. Most of them are generally available, for instance PKZIP, ARC, LHARC or ZOO. The last mentioned, ZOO, was choosen as the starting point for the development of RZOO. This archive program, developed by Rahul Dhesi, has very effective compression algorithms.[1] It is a highly portable program that is already running on many platforms, including MSDOS environments. Moreover, Rahul Dhesi who has the copyright of this program has made executable versions as well as the program's source publicly available without any licence fees.[2] It was, therefore, possible to get the source of the decompression routines of his archive program and to use them for the development of RZOO.

RZOO has the following main features:

- The program can operate directly on compressed data archives. No separate data decompression is needed.
- As a consequence, storage requirements and also disk access times to read the data are greatly reduced.

[1] For instance, the data file **datah**, a file of about 1 megabyte of numerical data taken from the SOEP data base, gives the following compression rates: 78% (ARC), 80% (LHARC), 80% (PKZIP), and 83% (ZOO).

[2] ZOO, as well as some other utilities working together with ZOO, can be found on many public network servers, for instance on the TRICKLE server which may be reached in Italy via the e-mail address TRICKLE @ IMIPOLI; it is also available on the mail server in Stuttgart (Germany) via the e-mail address MAIL-SERVER @ RUSINFO.RUS.UNI-STUTTGART.DE.

- All data can be accessed by variable names. There is almost no need to lern a specific retrieval language.
- The program offers the most needed possibilities to combine different record types, i.e. records form several different data files in the archive. In addition, the program can be used for external record matching.
- The program offers a flexible option to select records.

The following sections describe how to use RZOO und how the program works. Some small data files are used to illustrate the explanations.

2 Setup of an Archive

This section describes how to setup an archive to be used with RZOO. This is done in three steps.

- First, the user has to create a ZOO archive of his data files.
- Second, he/she must create a description of the variables to be found in the archive. Optionally, a dictionary file with value labels can be added.
- Finally, he/she has to write one or more small text files that describe the contents and operation of the archive.

Building the ZOO Archive

Assume there are n data files `dat.1, dat.2, ..., dat.n`. A ZOO archive, called `dat.zoo` in this example, is simply created by using the command

```
zoo ah dat.zoo dat.i
```

for each of the data files ($i = 1, \ldots, n$). The first command creates the archive file `dat.zoo` and adds the file `dat.1`. The following commands add the other data files. Note, that ZOO (version 2.1) can do the compression of files with two different algorithms. By appending `h` to the command option `a` (add a file to the archive) one chooses the second algorithms which gives a somewhat higher compression rate and shorter decompression times. Although RZOO supports both compression algorithms of ZOO, it is highly recommened to use the second one that is always more efficient.

To be sure, there are a lot of other commands that can be used with ZOO. The program provides for all that is needed for a full manipulation of data archives, including some possibilities to repair damaged archives.

As an example, used througout this text, we take the following four data files:

```
   dat.1          dat.2          dat.3       dat.p
----------     -----------     ---------   -------
1 -1.01 90     1 1 1.01 -1     90 9001 1     1 90
6 -1.06 95     2 6 1.06 -2     91 9003 3     2 90
2 -1.02 90     3 2 1.02 -3     95 9008 8     3 91
7 -1.07 95     4 7       -4    97 9009 9     4 91
3 -1.03 91     5 3 1.03 -5                   5 91
8 -1.08 95     6 8 1.08                      6 95
4 -1.04 91     744 1.04-77                   7 95
9 -1.09 97     8 9       -8                  8 95
5 -1.10 91                                   9 97
```

We will call the ZOO archive **exam.zoo**, so it is made by `zoo ah exam.zoo dat.i` for all four files. The result can be investigated with the ZOO program. For example, the command `zoo l exam` results in:

```
Archive exam.zoo:
Length    CF  Size Now  Date      Time
--------  --- -------- --------- --------
     108  48%       56  29 Jan 92 19:22:56     dat.1
      96  35%       62  29 Jan 92 19:23:32     dat.2
      40  20%       32  29 Jan 92 19:23:58     dat.3
      45  27%       33  29 Jan 92 19:24:38     dat.p
--------  --- -------- --------- --------
     289  37%      183     4 files
```

It is important that all files in an archive to be processed by RZOO have a record structure. The program must be able to distinguish different records. Two types of files can be used: files with fixed and variable length records. With the first type of files it is assumed that all records have the same length (to be defined in the archive description file, see below). In this case an end-of-record character is optional. With the second type of files the records may have variable length. In this case an end-of-record character is obligatory.

Unfortunately, the end-of-record, also called newline character is not standardized between different platforms. On UNIX machines it is a single character, normally a

linefeed (hex 0a). On the MACINTOSH it is a single carriage return (hex 0d). DOS makes files a little bit longer by taking two characters for the end of a record: carriage return and linefeed. All three possibilities may be used for files with variable length records contained in an archive. However, when the program *writes* an end-of-record (newline) character it is always the appropriate one for the target machine where the program is running.

Describing the Variables

The next step is to create a description of all the variables to be found in the archive. This is necessary because the intention is to retrieve the archive data by variable names. Therefore, we a description of the variables is needed, and where to find them.

This is made available to RZOO by the creation of a *variable description file*. After it is created, it should be added to the ZOO archive as described in the first step.

A variable description file is a simple text file that can be created with the help of any text editor. Each line can be a comment line, or it is the description of one variable. A line of the file is interpreted as a comment line if it is empty or if its first non-blank character is an asterix (*) or a pound sign (#). Any other line is taken to contain a variable description.

A line with a variable description consists of five fields, the last field with a variable label is optional. Fields are taken to be in a free format, separated by at least one blank character.

1. The first field is the *name of the variable*. The number of characters of a variable's name is not fixed. It is proposed, however, that a maximum of eight characters is used. The characters can be uppercase and lowercase letters, digits, and the underline character. The first character must be a letter.

2. The second field must be a numerical entry and is interpreted as the *ID number of the data file* where the variable is to be found.

3. The third field is again numerical and is interpreted as an *offset in the data file record* were the value of the variable begins. Note that counting begins with zero as the first column in a data file's record.

4. The fourth field is also numerical and is interpreted as the *length of the field* that is occupated by the variable. It should be, for instance, the number 5 if

the variable occupates five columns in the data file's record. Note that it is not necessary to give specific information about the type of the variable. It can be an integer or a floating point value. The interpretation is done automatically by RZOO.

5. The last field is optional and can be an arbitrary string of characters that describes the variable as a *variable label*. Different from the other items, this field may contain blank characters. If it is present it starts with the first non-blank character after the fourth field up to the end of the current line.

To illustrate the setup of a variable description file, we take again the example data introduced above. We will assume that the files contain three, four, three, and two variables, respectively, defined by the following variable description file:

```
*  var.des   Variable description file for exam.zoo
*  Name     File   Offset   Length    Label
   Var_A1     1        0        2     1. variable of dat.1
   Var_A2     1        3        5     2. variable of dat.1
   Var_A3     1        9        2     3. variable of dat.1

   Var_B1     2        0        1     1. variable of dat.2
   Var_B2     2        1        2     2. variable of dat.2
   Var_B3     2        4        4     3. variable of dat.2
   Var_B4     2        8        3     4. variable of dat.2

   Var_C1     3        0        2     1. variable of dat.2
   Var_C2     3        3        4     2. variable of dat.2
   Var_C3     3        7        2     3. variable of dat.2

   Var_P1     4        0        1     1. variable of dat.p
   Var_P2     4        1        3     2. variable of dat.p
```

Adding this file to **exam.zoo** completes the basic version of a ZOO archive to be used with RZOO.

Note: A single variable description file is used to describe *all* variables in an archive. It is not required that all variables that are actually contained in an archive are explicitly described. However, only variables that have a description can be retrieved. The ordering of the variable descriptions is not important.

Information about Value Labels

A variable description files has information about the names of available variables in the archive, optionally supplemented by variable labels. Sometimes it is helpful to have also value labels, i.e. information about the meaning of different values of the variables.

As an option, such information can be put in an archive to be used by RZOO in the form of a *dictionary file*. A dictionary file is created similar to a variable description file, the following is an example:

```
* dic. Example of a dictionary file for exam.zoo

Var_A1    (1) value 1
          (2) value 2
          (3) ...
Var_A3    No information about labels
Var_P1
          Label 1
          Label 2
Var_P2    Used as a pointer variable
```

A dictionary file is an ordinary text file, normally with variable length records. It may contain comment lines in the same way as a variable description file, all other lines are interpreted. When a line starts with a string it is interpreted as the name of a variable. All following lines with at least one blank (or tab) character at the beginning of the line, up to the next variable name (or up to the end of the file) are taken to be value labels associated with the variable. The line where the name of the variable is given may also contain a value label. This is optional.

The interpretation of a dictionary file by RZOO is based on the following rules:

1. Variable names must always start at the beginning of a line, whenever a line begins with at least one blank or tab character it is interpreted as a value label.

2. A value label starts with the first non-blank or non-tab character in a line, or after a variable name, and extend until the end of the current line.

3. All variable names contained in a dictionary file must also be contained in the variable description file. However, variables that are contained in the variable description file may be omitted in the associated dictionary file.

4. Most important, the ordering of the variables in the dictionary file must be the same as the ordering in the variable description file.

Describing the Archive Structure

In the third step, it is necessary to supply RZOO with some additional information about the archive. This is done by creating one ore more *archive description files*. Again, these are simple text files that can be created with the help of any editor.

Archive description files can have comment lines and information lines, in the same way as variable description files. Each line is interpreted as a comment line if it is empty or if its first non-blank character is an asterix (*) or a pound sign (#). All other lines are regarded as information lines and interpreted by RZOO.

The entry in the first information line in an archive description file is always interpreted as the name of the ZOO archive that is to be described by the file. The archive name is taken from this line starting with the first non-blank character up to the next blank character. Of course, it is possible to define a complete path to the archive file in this line.

All other information lines in an archive description file are interpreted as descriptions of data files contained in the archive. Each of these lines may have up to eight entries, the first six entries are obligatory, the last two entries are optional. Again, each entry is taken as a field of non-blank characters, separated by one or more blanks.

1. The first field is the *ID number*, or *logical number*, *of the data file* described in this line of the archive description file. It must be a non-negative integer number, unique for each file described in the file. The ordering is not important. This number is taken as the ID number of the data file, and it must be the same number as used for attaching variable names to data files.

2. The second field is the *name of the file* that is described in this information line. It must be the same name as used in the creation of the ZOO archive. Note: file names may be built in accordance with the operating system's requirements. However, the printout relies on the DOS convention that file names have a maximum of 12 characters.

3. The third field is a numerical entry and is interpreted as the *type of the file*. In the current program version, three values are allowed: If the value is 1

the file is assumed to be a data file. If the value is 2 the file is taken to be the variable description file of the archive. A file of type 3 is interpreted as a dictionary file. There may be only one variable description file and, optionally, one dictionary file. The number of data files is only restricted by some limits defined at compile time of the program.

4. The fourth field is a numerical entry and is interpreted as the *record length of the file*. It must be a non-negative integer. If its value is greater than zero it is assumed that the file consists of records of equal length given by this value. (Note, if the records have one or more end-of-line characters they should be included in the record length.) If on the other hand the value is zero, it is assumed that the file has variable length records. In addition, it is then assumed that each record is terminated by an end-of-line character. Normally, data files will have fixed length records, while variable description files will have variable length records. However, also data files may have variable length records, if only the variables that are described in the variable description file can be found at fixed positions in the records.

5. The fifth field is a numerical entry and is interpreted as the *number of records of the file*.

6. The sixth field is a numerical entry and is interpreted as the *number of variables* contained in the file. Currently, this value is used for informational purposes only. In the case of a variable description or dictionary file this entry should be set to zero.

7. The following two entries are optional. The seventh entry is interpreted as the *name of a matching variable*. If it is defined it must be a variable contained in the data file described in this line of the archive desription file. The purpose of this entry is to allow for a matching of records of different data files. The process of matching is described in detail in Section 4.

8. The last entry is again interpreted as the *name of a pointer variable* to be used for matching records from different data files. The meaning is also described in detail in Section 4.

The following, `exam.1`, is an example of an archive description file for the archive `exam.zoo`. In this first example matching or pointer variables are not used.

```
* exam.1
* RZoo Description file for exam.zoo

  exam.zoo

* N  Name     Type  RecLen  Records    NVar  Match  Pointer
  1  dat.1       1      12        9       3
  2  dat.2       1      12        8       4
  3  dat.3       1      10        4       3
  4  dat.p       1       5        9       2
  8  var.des     2       0       18       0
  9  dic         3       0       10       0
```

Empty lines, and lines starting with an asterix, are comment lines. The first information line contains the name (optionally the path) of the ZOO archive. In this example it is `exam.zoo`. The following information lines describe the files contained in the archive. The ordering is not important. In this example, there are four data files of type 1, one variable description file of type 2, and a dictionary file of type 3. The data files are defined with a fixed record length, the variable description file and the dictionary file have variable length records.

Note: The definition of record length, as used by RZOO, always includes any end-of-record characters. This may result in differences between DOS and UNIX systems. The above example was produced on an UNIX system where only one character is used for marking the end of a record in a text file. However, archives may be transferred *without any modification* (as binary files) between UNIX and DOS systems, and then the description of the archive is also the same on both systems.

3 Description of Commands

Before proceeding with an explanation of the operation of RZOO, this section describes the commands that can be used to control the operation of the program. Commands may be given on the command line, when invoking the program, and/or through the means of a *command file* as described in the `cf`-command. If the pro-

gram is invoked without any commands it prints a small usage message on the standard output of the system.[3]

`d=` This command must always be used to define the name (and optionally the path) of the archive description file.

`test` This command may be used to check the archives integrity. If this command is given RZOO reads, in turn, all files described in the archive description file. If inconsistencies are found with regard to the number and/or length of record definitions the user is notified on the standard output. In addition, if matching variables are defined, the program checks if they are in ascending order. This is required for the matching of records without sorting. After completion of this command the program is exited.

`dexp` For reading numerical entries in data files two different options are available. Firstly, a relatively fast routine that can interpret integer values and floating point values in the *F*-format. This routine is used as a default. The `dexp`-command activates the alternative routine, that is the `sscanf` function contained in almost all C compiler standard libraries. This function is somewhat slower. However, it has the advantage that it can additionally interpret floating point numbers in the *E*-format. If such a format is not used, both routines should give the same result.

`var`
`vvar` These commands may be used to print the description of variables defined in the variable description file on the standard output. The first command, `var`, gives an unmodified printout of the contents of the variable description file. The second command, `vvar`, prefixes each line with an asterix (*) and inserts a semicolon after each variable's name. The result may then be used for the creation of a command file. Both commands may also be used in the following forms: `var = n` or `var = fname`. Then only variables form the data file with ID number `n` , or file name `fname`, are printed.

[3]The standard output is normally the screen of a terminal; however, it may be redirected to any file with the help of the > or >> operators provided by the operating system.

`dic` When this command is used the program tries to add information about value labes to the description of variables. It may be used in addition to the `var` or `vvar` commands, or when retrieving data. Of course, there must be a dictionary file in the archive, otherwise the `dic` command is ignored.

`v=` This command may be used to define the names of variables to be retrieved. For each variable a separate `v = vname` command should be used. The limit is currently set to 500 but may be changed when the program is compiled. The ordering of the variable names can be choosen freely by the user. The program uses the same ordering in the creation of the output file.

Note: The first variable defined with the `v`-command, called the *primary variable*, has a special significance. The output data file is always organized according to the data file that contains this primary variable. If not restricted by the `noc`-command, the number of records of the output data file is the same as the number of records in the data file the primary variable belongs to. The ordering of the records is the same as in this data file, unless a sort of the output data file is requested. Moreover, if the output data file is sorted it is always done in ascending order with respect to the primary variable.

`df=` This command should be used to define the name of an output data file. There are two options. With the command `df = 1` the retrieved data are written to the standard output of the system. With the command `df = fname` the program creates a file with the name `fname` and writes the retrieved data into this file. Note that any already existing file with the same name is overwritten without warning. If variables are requested but no output file is defined the program gives a message and exits without retrieving any data. However, in any case a description of the variables to be retrieved is printed.

`free` Retrieved data are printed into the output data file in two different ways. As a default the data are written exactly the same way as found in the archive data files, the variable fields, as defined by the variable description file, are copied without any modification to the output data file. In particular, no recoding of missing values is done.

If the `free`-command is specified, the variable data are first translated to internal floating point values (using the default or `sscanf` routine, depending on the `dexp`-command), and then written to the output data file in a free format, i.e. printed in a field as small as possible. Floating point values are printed in a field of maximal eight characters. Missing values are recoded as is explained below.

`sep=` This command can be used to define a separation character that is printed between the value fields in the output data file. As a default, the separation character is taken as a blank. If `sep = none` is specified no separation character is printed.

`noc=` As a default the number of records written into the output data file is the same as the number of records found in the primary data file, i.e. the data file to which the primary variable belongs (see above). With the command `noc = n` (any positive integer) only `n` records are written. The command is also recognized with the `pdf` (print data file) option and with external record matching.

`sel=` This command may be used to select records to be written into the output file. The syntax is `sel = op(V1,V2,...)` where `V1, V2, ...` are variables as defined in the variable description file. `op` is an operator that produces a numerical value from the variables. The meaning is that only records are selected for which the resulting numerical value is not equal to zero. The available operators are described in detail in the next section. The `sel`-command may be used several times. The connection is then taken to be a logical OR, i.e. records are selected if at least one of the select strings results in a value different from zero.

sel()=

The **sel**-command, described above, is active while writing the output file. Since the selection of records requires that record matching is already finished, it can not be done in an earlier stage. However, the program may be instructed to select records while reading data files from the archive. This option is specified by adding a file ID number or a file name to the **sel**-command. The syntax is: **sel(n) = op(V1,V2,...)** where **n** is a file ID number, or **sel(fname) = op(V1,V2,...)** where **fname** is the name of a data file. The interpretation is the same as described above. Of course, the variables used in a definition must belong to the specified file since, while reading, information about other record types is not available.

snm3

This command may be used to select records without missing values of type 3. For each record of the primary file there must be a matching record of all other files requested, otherwise the record is not printed into the output file.

osort

As a default the ordering of the records in the output data file is the same as in the primary data file with respect to the primary variable. If the **osort**-command is specified the records in the output data file are sorted in ascending order of the primary variable.

Note: Sorting is expensive since the data must first be written into a temporary file. Only pointers to the records of the temporary file are sorted, the data of the temporary file are then read again and written into the output data file.

dsort
nsort

If variables from more than one data file are to be retrieved simultaneously some way of reaching a match of the records of the different data file is needed. As explained below, this is done by using matching variables. There are two possibilities. If the data files are already sorted (in ascending order) with respect to the matching variables, matching of records can be done without any further sorting. In this case the **nsort**-(no sort) command is appropriate.

If, however, the data files are not sorted with respect to the matching variables they must be sorted by the program before any match of records can be reached. In this case the **dsort-** (do sorting) command is appropriate. The data are then, in a first step, written into a temporary file, in a second step pointers to the records of this temporary file are sorted, and finally it is tried to match the records and the matching records are written into the output data file.

The default of these commands depends on the type of retrieval. If all variables to be retrieved belong to a single data file the default is **nsort**, otherwise the default is **dsort**.

keep

When record matching requires the data file records to be sorted, the program creates a temporary file called RZTMP001. A second temporary file, called RZDIC001, is created when information about value labes have to be retrieved from a dictionary file. As a default, temporary files are deleted when they are no longer needed. If the **keep** command is given, the temporary files are not deleted and may be inspected or otherwise used after termination of the program.

mvc1=
mvc2=
mvc3=

The program distinguishes three types of missing values. The missing value type 3 is used when no matching record that contains the variable can be found. The missing value type 2 is used when the field of a variable consists entirely of blank characters. The missing value type 1 is used when the field of a variable can not be interpreted as a numerical value but no other missing value type applies. This may be due, for example, to non-numerical entries in the field of a variable.

The **mvc1 =** , **mvc2 =** , and **mvc3 =** commands may be used to change the three default missing value codes which are -1, -2, and -3, respectively.

Note, however, that these missing value codes are active only if the output data file is writte in a free format. Otherwise, as a default, the variable fields are copied without any modification into the output file, and missing values of type 3 are transformed into blank fields.

mvc9= Sometimes missing values are coded by filling the whole field of a variable with nines. As a default, this is not recognized by RZOO. However, this may be changed with the command **mvc9 = d** where d is an arbitray floating point value. Then, whenever the program finds that the field of a variable is filled with nines, it substitutes the value **d**.

spss= If the command **spss = fname** is given, with **fname** any file name, the program creates a file with this name and prints names and labels of the variables to be retrieved into this file. This is done according to the syntax of SPSS, so the file may be used to generate SPSS procedures.

pdf= This command may be used to print the contents of a file from the archive on standard output. The syntax is: **pdf = n** where **n** is the ID number of the file, or **pdf = fname** where **fname** is the name of the file. As a default, the file is printed to the standard output without any modification. This should give the same result as using the extract option of ZOO. The **noc**-command may be used to limit the number of records written. The **eol**-command may be used to create end-of-line characters.

eol As a default, the file requested with the **pdf**-command is written without any modification. If the **eol**-command is used, each record is supplied with an end-of-line character. If the data file have already an end-of-line character, i.e. if it is defined with a zero record length in the archive description, it is substituted by an end-of-line character appropriate for the target machine where the program is running. On DOS machines this results in two characters: carriage return and linefeed.

`mf=` This command my be used to perform an external record matching. At least two files have to be defined. The syntax is: `mf = n,c,fname` or `mf = n,off,len,fname` depending on the type of the file. In both cases, `fname` is the name of the file and `n` is an upper limit for the number of records to be read from the file. Record matching is done by sorting each of the files with respect to a matching variable, defined by `c` or `off,len`. If the file has a free format, the first form must be taken where `c` is the number of the variable in the file (counting the variables in each record by: $1, 2, 3, \ldots$). If the file has a fixed format, the matching variable has to be defined by an offset (`off`) and a length (`len`).

The first file defined with the `mf`-command is the reference. Matching records are written to the standard output in a single line for each set of records from the input files. The records are separated by a separation character as defined by the `sep`-command. The default is a blank character.

Note: Regardless of their type, free or fixed format, it is always assumed that files used for external record matching consist of records delimited by an end-of-record character, appropriate for the machine where the program is running.

`cf=` Finally, with this command the name of a command file can be given to the program. It is a standard text file containing any series of commands in an arbitrary order. A command may contain blanks and may be longer than one line. However, every command must be finished by a semicolon. All blanks are ignored. Also entries following a semicolon on the same line are ignored and may be used for comments. Any empty lines, and lines with an asterix (*) or pound sign (#) as first non-blank character, are ignored as comment lines.

4 Operation of the Program

This section explains the operation of RZOO. To illustrate the explanations, the example data introduced in Section 2 are used.

If the program is invoked without any command, that is if only `rzoo` is typed, the program gives a short message about its possible usage and then exits. To do anything else, the program needs at least an archive description file. This information is given to the program by the `d = name` command, where `name` is the name of the archive description file to be used.

If at least this command is given to RZOO, the program tries to open this file and to interpret its contents. It looks for the definition of the ZOO archive file, reads directories of the archive, and examines if all files, defined in the archive description file, are contained in the archive. In addition, it is checked if there is exactly one variable description file. If one of these checks fails the program exits with an error message.

If the program is invoked with *only* the definition of an archive description file, it prints its interpretation of this file to the standard output of the system. In our example, if we type `rzoo d=exam.1`, we will get the following output:

```
Archive description file: exam.1
Archive: exam.zoo

FN  Typ RLen Records NVar   Match    Pointer    Size    M  Name
--- --- ---- ------- ---- --------- --------- --------  -  ----
  1   1    12       9    3                        108  2  dat.1
  2   1    12       8    4                         96  2  dat.2
  3   1    10       4    3                         40  2  dat.3
  4   1     5       9    2                         45  2  dat.p
  8   2     0      18    0                        795  2  var.des
  9   3     0      10    0                        214  2  dic
```

Most entries are the same as defined in the archive description file. `ID` is the ID number of the file. `Size` is the *uncompressed* size of the file (in bytes), as found in the archive. The fields `Match` and `Pointer` are empty since no matching and pointer variables are used in this example. `M` is an information about the compression method used in the archive. There are three possible methods: (0) if the file is not compressed, (1) if the standard compression method of ZOO is used, (2) if the optimized compression method is used. This last method should always be used when creating archives for RZOO. It is achieved by using the `h` flag of ZOO.

Now, after this preliminary information, there are basically the following different usages of the program:

- To retrieve files from the archive.
- To get information about available variables.
- To check the archive and its description.
- To retrieve variables from the archive.
- To perform external record matching.

4.1 Retrieving Individual Files

To retrieve individual files from an archive one has to use the command `pdf = n` where `n` is the ID number of a file defined in the archive description file, or `pdf = fname` where `fname` is the name of the file. The file is then printed without any modification to the standard output of the system. Of course, one can do the same with ZOO, and the result should be the same.

Two additional options are available. The number of records written to the output can be restricted with the `noc`-command. As a default, all records found in the archive file are written. The command `noc = n` limits the output to a maximum of `n` records. This may be useful, for example, to have a short look to the first records of a file.

Another option is provided by the `eol`-command. An end-of-record (newline) character is then written at the end of each record. This is a useful option if newline characters are not already included, or if such characters are generated on a different machine. The newline characters written by RZOO are always appropriate for the machine where the program is running. Therefore, it should be possible to edit the output on the target machine regardless of where the archive was created.

As an example, the command

```
rzoo d=exam.1 pdf=dat.2 noc=3
```

produces the following output:

```
Contents of data file: dat.2 (2)

1 1 1.01 -1
2 6 1.06 -2
3 2 1.02 -3

Number of records: 3
Memory used: 30028 bytes.
```

The number in brackets, following the name of the file, is its ID number as defined in the archive description file. The number of records is printed only if the `noc` or the `eol` command is used. As a default, the file is printed without any modification and records are not counted.

4.2 Information about Available Variables

To get information about available variables, the commands `var` and `vvar` can be used. The program then prints the variables and their descriptions, as found in the variable description file, to the standard output of the system. If the second form, `vvar`, is used, each line is preceeded by the string `* v =` and a semicolong is inserted after each variable name. This makes the output suitable for an immediate command file creation.

It is possible to request only descriptions of variables for a specific data file. This may be done by adding an ID number or the name of a file to the `var` or `vvar` command. For example, the command

```
rzoo d=exam.1 vvar=dat.1
```

gives the result:

```
Available variables: dat.1 (1)

* v = Var_A1;    1        0        2     1. variable of dat.1
* v = Var_A2;    1        2        5     2. variable of dat.1
* v = Var_A3;    1        7        2     3. variable of dat.1

Number of variables: 3
```

As a default, variable descriptions are printed without information about value labels. If such information is available in the archive, it may be requested by adding

the `dic` command. For example, adding `dic` to the just mentioned command line results in:

```
Available variables: dat.1 (1)

* v = Var_A1;     1         0         2      1. variable of dat.1
*    (1) value 1
*    (2) value 2
*    (3) ...
* v = Var_A2;     1         2         5      2. variable of dat.1
* v = Var_A3;     1         7         2      3. variable of dat.1
*    No information about labels

Number of variables: 3
```

Note: Lines from the variable description file and from the dictionary file are printed without any modification, only leading blank and tab characters are skipped. A sensible layout should be choosen when making these files.

4.3 Test of the Archive Setup

A third option is to test the integrity of the archive setup. This is done with the `test`-command. The program then does the following: It first tries to read all files defined in the archive description file and counts the number of records in the files. The number of records actually found is printed to the standard output and, if there are discrepancies between these counts and the definitions in the archive description file, one or two error messages are printed.

The first error message (code 1) signifies that the program found a discrepancy between the number of records defined in the archive description file and the number of records actually found in the file. The second error message (code 2) is given when there seems to be an inconsistency between the defined and actually found record length. However, the reason may not be unique. This is due to the method used by the program to read single records from the files. It is always done in one of two different ways: If the record length of a file, as defined in the archive description file, is zero, then the program assumes that each record of the file has an end-of-line character. This end-of-line character (on DOS systems it may be two characters) is used to determine the actual length of the record. However, if the record length is

defined in the archive description file by a positive number, the program assumes that *all* records of the file have the same record length of exactly this number of characters. Any eventually existing end-of-line characters are ignored and treated as part of the record.

An additional check is done if matching variables are defined in the archive description file. For each file where a matching variable is defined, it is then checked if the file is sorted in ascending order with respect to this variable. The result of this check is also reported on the standard output.

As an example, the command

```
rzoo d=exam.1 test
```

results in the following output:

```
Test of archive and description file

FN   0  File: dat.1          Records:       9  Err 0
FN   1  File: dat.2          Records:       8  Err 0
FN   2  File: dat.3          Records:       4  Err 0
FN   3  File: dat.p          Records:       9  Err 0
FN   4  File: var.des        Records:      18  Err 0
FN   5  File: dic            Records:      10  Err 0
```

4.4 Retrieving Variables from the Archive

This, of course, is the main usage of RZOO and why it was developed. Basically, one has to distinguish two different situations. First, if a set of variables to be retrieved belongs to only a single data file in the archive, the operation of the program is very simple. It only has to read the data file and get the requested variables to be printed on the standard output or into an output file. If, however, the variables belong to two or more different data files, the operation is more complicated. The basic problem then is how to match the records from the different files.

In both cases, to retrieve any variables, the user must at least define the requested variables, with the `v`-command, and specify an output file, with the `df`-command, where the retrieved data should be written. In addition, several other commands may be used, they are described below.

Single Data Files

To explain the operation of the program in this case we consider the task of retrieving all variables from the first of the above described example data files. This may be done with the following command file:

```
* rr.1  Example command file

    d = exam.1;      archive description file
    df = 1;          print to standard output
    v = Var_A1;      first (primary) variable
    v = Var_A3;      third variable
    v = Var_A2;      second variable
```

This example command file, called `rr.1`, is simply used by invoking RZOO as

```
rzoo cf=rr.1
```

This resulting output should be:

```
Archive description file: exam.1
Archive: exam.zoo
Command file: rr.1

Selected variables

Var     Name         File       Offs   Len  Label
---  ----------  ------------  ------  ---  -----
  1  Var_A1      dat.1              0    2  1. variable of dat.1
  2  Var_A3      dat.1              7    2  3. variable of dat.1
  3  Var_A2      dat.1              2    5  2. variable of dat.1
```

```
Primary file: dat.1 (1)

1 90 -1.01
6 95 -1.06
2 90 -1.02
7 95 -1.07
3 91 -1.03
8 95 -1.08
4 91 -1.04
9 97 -1.09
5 91 -1.10

Number of missing values
Var     Name       Type 1   Type 2   Type 3    Total
---  ----------  -------  -------  -------  --------
  1  Var_A1           -        -        0         0
  2  Var_A3           -        -        0         0
  3  Var_A2           -        -        0         0

Number of records: 9
Memory used: 34102 bytes.
```

To produce this output, the program has done the following things: First, it has tried to find all requested variables in the variable description file. If a variable could not be found the program gives an error message and exits. Then the ID numbers of the files connected with the requested variables are determined, also were the variables are to be found in the data files. If all variables belong to the same data file the problem of matching records is skipped. The requested data file is read from the archive, the requested variables are taken from its records and printed to the specified output channel.

Sorting the Output Data

As a consequence of this procedure, the ordering of the records in the output file is the same as in the retrieved data file. This results in a very fast operation. Things are more complicated if the user wishes the output file to be sorted, using the `osort` option. Since it cannot be assumed that all data can be stored in internal memory, the program writes the requested data to a temporary file (always called `RZTMP001`)

and recognizes only pointers to the records of this file. Then these pointers are sorted. Finally, the temporary file is read again, according to these sorted pointers, and the sorted records are then written into the output channel.

Note, that in the case of a single data file a sort of the output data, if requested, is always done with respect to the *primary variable*, that is the first of the requested variables. This is different to the case of two or more data files where the sort is done with respect to the matching variables.

Note also, that currently sorting is only possible with integer valued variables. Floating point values are, without warning, casted into integer values before any sort is done.

Options for the Print Format

Another aspect of the output file is how the data are printed. Two options are available. First, as a default, the data are written in exactly the same way as they are found in the archive data files. This is done without any interpretation of the data as numerical variables (so there is an indirect possibility to handle string variables in this case), they are simply copied from the fields where they are defined to the output data file.

As another option the user may specify the `free`-command. The effect is that the variables to be retrieved are first interpreted as numerical values before they are written to the output file. They are then written in a so-called *free format*, for each value the smallest possible amount of places is used.

With both options the user has the possibility to specify a character to separate the value fields in the output file. The default is a blank character. Any other character may be defined with the `sep`-command. If the user specifies `sep = none` the fields are not separated.

Missing Values

RZOO distinguishes between three types of missing values. The type 3 is assumed when no matching record can be found; this is explained below. The type 2 is assumed when the field of a variable consists entirely of blank characters. The type 1 is assumed when no other type applies but the field of the variable cannot be interpreted as a numerical value.

The representation of missing values in the output file depends on the printout format. If the default format is choosen missing values have the same representation as found in the archive data files. In this case each variable's field is copied directly and without any modification to the output file, missing values of type 3 are printed as blank fields. If, instead, the free format is choosen, the program substitutes missing values by codes that can be specified by the user, cf. the `mvci`-commands. The default values are -1, -2, and -3, respectively.

In addition, the program can be instructed to recognize fields of variables that consist entirely of nines (9) to be missing values. This is not done by default, it must be specified with the `mvc9 = code` command. Then `code` is substituted for this type of missing values. Note again, that also this option is only active if the free format command is used for writing the output.

Select Cases Options

Two options for a selection of records are available. A trivial one by using the `noc = n` command. Then only the first `n` records are printed to the output file. Note, however, that the restriction by the `noc = n` command is recognized while *reading* the archive data files. If a sort of the output data is requested one will not get the first `n` sorted records, but a sort of the first `n` records found in the data file.

A more useful select cases option is supplied by the `sel`-command. The general syntax is `sel = op (V1, V2, ...)` where `V1, V2, ...` are variables that *must* be contained in the set of requested variables defined by the `v`-command. `op` is a simple or complex operator that transforms the variables into a single numerical value. A record is selected if the result of this operator is a non-zero value. Otherwise, the record is skipped and not printed into the output file. If more than one of these select commands are used they are connected by a logical OR: records are selected if at least one of the operators results in a non-zero value.

There are several different operators, and they may be freely used in complex expressions. Brackets may be used to control the priority of evaluation. First, all elementary operators are available: `+, -, *, /`. In addition there is a logical multiplication – each non-zero result is evaluated to one –, that is a shortform for the logical AND, defined by a single point.

The other available operators are:

`INT(V)`	Integer value of V
`ABS(V)`	Absolute value of V
`NOT(V)`	Logical negation of V
`ET(V1,V2)`	Logical conjunction of V_1 and V_2
`VEL(V1,V2)`	Logical disjunction of V_1 and V_2
`AUT(V1,V2)`	Exclusive disjunction of V_1 and V_2
`EQ(V1,V2)`	Result 1, if $V_1 = V_2$, else 0
`NE(V1,V2)`	Result 1, if $V_1 \neq V_2$, else 0
`LT(V1,V2)`	Result 1, if $V_1 < V_2$, else 0
`LE(V1,V2)`	Result 1, if $V_1 \leq V_2$, else 0
`GT(V1,V2)`	Result 1, if $V_1 > V_2$, else 0
`GE(V1,V2)`	Result 1, if $V_1 \geq V_3$, else 0
`IF(V1,V2,V3)`	Result V_2, if $V_1 \neq 0$, else V_3

For example: the command `sel = gt(Var_A1 + Var_A3, 100)` should give a selection of all records where the sum of the first and the third variable is greater than 100. And the command `sel = not (gt(Var_A1 + Var_A3, 100))` should give all other records.

Two or More Data Files

If the requested variables belong to two or more data files, the main problem is how to match the records from the files. RZOO provides two different methods: a simple match via matching variables, and a more complicated matching procedure by using pointer variables. In the following we describe both methods and give some illustrations. Most options, as explained above, carry almost without any modification over to the case of two or more data files. So they are not explained again.

Matching Variables

The idea of matching variables is very simple. Assume that the variables to be retrieved belong to two data files. We further assume that there is, for each of the files, exactly one variable – called a *matching variable* – that can be used to match records in the following sense: two records match together if the matching variables have the same value in each record. The idea is analogous for three or more data files.

Obviously, in many cases different possibilities to define matching variables exist. First, different variables can be selected to become matching variables. This is in

the responsibility of the user of RZOO. The best way is to define some prototypes of matching, appropriate for a given archive, and then to setup a specific archive description file for each prototype. Second, the process of matching via matching variables may not be unique if the matching variables have the same values in two ore more different records. Again, this possibility is not excluded by RZOO.

As done by RZOO, the matching of records is based on the concept of a *primary matching variable*. This is the matching variable defined for the primary data file (implicitly specified by the first variable requested). The values of the primary matching variable are the reference values. All other data files are checked in turn if at least one matching record can be found. Clearly, if two or more records have the same value of the matching variable the selection is not unique.

Another problem is how to make the match in an efficient way. There are two different situations. One situation arises when the data files are already sorted in the same order with respect to their matching variables.[4] Then the process of matching records can be done without any further sorting while reading the data files from the archive. This is the most efficient way. If, however, the data files are not already sorted with respect to their matching variables, sorting them should be done before the process of matching is tried. Therefore, in this case, the program creates a temporary file, writes all requested data into this temporary file, then makes a sort of pointers to the records of this file, and finally tries to match the records.

The selection of these methods can be controlled by the user with the `nsort`- and `dsort`-commands. With the `nsort`-command the program assumes that the data files are already sorted with respect to their matching variables.[5] With the `dsort`-command the data files are always sorted before any matching of records is tried. Note, that the `osort`-command implies a `dsort`. If no sorting option is explicitly specified the program selects its way according to the number of data files needed. If only one data file is used in the retrieval, no sort is done, otherwise all data files, as far as they are needed, are sorted with respect to their matching variables.

We now give an illustration with the example archive described above. We will assume that the matching variables are `Var_A1`, `Var_B2`, `Var_C3`, and `Var_P1`. The first step then is to create a new archive description file that contains the information about these matching variables. Of course, the file `exam.1` can be used, one only has to add the names of the matching variables in the appropriate fields. We will call this second archive description file `exam.2`. It looks as follows:

[4]Sorts are always interpreted to be in ascending order.

[5]This can be checked with the `test`-command, see above.

```
* exam.2
* RZoo Description file for exam.zoo
* with some matching variables

  exam.zoo

* N  Name     Type  RecLen  Records    NVar  Match   Pointer
  1  dat.1      1      12        9       3  Var_A1
  2  dat.2      1      12        8       4  Var_B2
  3  dat.3      1      10        4       3  Var_C3
  4  dat.p      1       5        9       2  Var_P1
  8  var.des    2       0       18       0
  9  dic        3       0       10       0
```

As an example, we take the task of retrieving variables from the first three data files. This may be accomplished with the following command file:

```
* rr.2  Example command file with record matching

    d = exam.2;      archive description file
    df = 1;          print to standard output
    v = Var_A1;      first (primary) variable
    v = Var_A3;
    v = Var_B1;
    v = Var_B2;
    v = Var_C3;
```

The task is to retrieve variables from three data files simultaneously. Since no sorting option is specified the program follows its default, in this case it sorts the files. The resulting output is:

```
Archive description file: exam.2
Archive: exam.zoo
Command file: rr.2

Selected variables

Var     Name          File        Offs   Len  Label
---  ----------  ------------  ------  ---  -----
  1  Var_A1      dat.1              0    2  1. variable of dat.1
  2  Var_A3      dat.1              9    2  3. variable of dat.1
  3  Var_B1      dat.2              0    1  1. variable of dat.2
  4  Var_B2      dat.2              1    2  2. variable of dat.2
  5  Var_C3      dat.3              7    2  3. variable of dat.2

Matching
   File: dat.1          Matching variable: Var_A1
   File: dat.2          Matching variable: Var_B2
   File: dat.3          Matching variable: Var_C3

Primary file: dat.1 (1)

 1 90 1  1  1
 6 95 2  6
 2 90 3  2
 7 95 4  7
 3 91 5  3  3
 8 95 6  8  8
 4 91
 9 97 8  9  9
 5 91
```

```
Number of missing values
Var      Name        Type 1   Type 2   Type 3    Total
---   ----------   -------  -------  -------  --------
  1   Var_A1           -        -          0         0
  2   Var_A3           -        -          0         0
  3   Var_B1           -        -          2         2
  4   Var_B2           -        -          2         2
  5   Var_C3           -        -          5         5

Number of records: 9
Memory used: 90210 bytes.
```

The retrieval follows the first data file, that is the primary file. Then it tries to find matching records from the other data files. This is not always successful. Therefore, some fields are missing. (This is type 3 of the missing value types; since the default printout format is choosen they are printed as blank fields.)

If the `osort` or the `nsort` command is added the result is as follows:

```
      osort                nsort
 ------------         ------------
 1 90 1  1  1         1 90 1  1  1
 2 90 3  2            6 95 2  6
 3 91 5  3  3         2 90
 4 91                 7 95
 5 91                 3 91
 6 95 2  6            8 95
 7 95 4  7            4 91
 8 95 6  8  8         9 97
 9 97 8  9  9         5 91
```

The result with the `osort`-command is essentially the same as before, only the order of the records in the output file has changed. The result of the `nsort`-command is senseless since it is produced on the false assumption that *all* requested data files are already sorted with respect to their matching variables.

Pointer Variables

A second method for matching records from different data files is through pointer variables. The method is similar to the use of matching variables, but it adds one

more level of reference. To use this method, there must be (at least) one data file in the archive that can serve for the linkage of pointers and matching variables. This file is called a *pointer file*. Whenever pointers are to be used for matching records the primary file must be the pointer file.

Now, for each data file that shall be used via pointers, there must be two reference variables. The first one is again a matching variable as explained above. The second variable has to belong not to the data file itself, but to the pointer (primary) file (which may, of course, be identical). The matching of records is then done the following way:

1. The records (and consequently the output file) are organized according to the primary variable. This variable always belongs to the pointer file.

2. For each other data file there must at least a matching variable. If there is only a matching variable, but no pointer variable, then the match is done between this matching variable and the primary variable (of the pointer file). This is essentially the same procedure as explained above.

3. If, however, in addition to a matching variable there is also a pointer variable defined for a data file, then the matching procedure is in two steps: First, the primary variable is matched with the pointer variable. (This is a trivial step since both variables must belong to the same file, the pointer file.) Second, the pointer variable is matched with the matching variable of the data file.

To illustrate using pointer variables, a third archive description file for `exam.zoo`, called `exam.3`, is taken as an example. It looks as follows:

```
* exam.3
* RZoo Description file for exam.zoo
* with matching and pointer variables

  exam.zoo

* N  Name     Type  RecLen  Records    NVar  Match    Pointer
  1  dat.1      1      12        9       3  Var_A1
  2  dat.2      1      12        8       4  Var_B2
  3  dat.3      1      10        4       3  Var_C1   Var_P2
  4  dat.p      1       5        9       2  Var_P1
  8  var.des    2       0       18       0
  9  dic        3       0       10       0
```

To support an interpretation, one may think of the variables Var_A1, Var_B2, Var_C3, and Var_P1 as ID numbers of individuals, and the variables Var_A3, Var_C1, and Var_P1 may be thought of as ID numbers of households. Then the problems is to match individuals and households. This can be done by taking dat.p as a pointer file since this file contains both individual and household IDs. To illustrate this matching we take the following command file:

```
* rr.3  Example command file with record matching
*       using matching and pointer variables

    d  = exam.3;     archive description file
    df = 1;          print to standard output
    v  = Var_P1;     first (primary) variable
    v  = Var_A1;
    v  = Var_A3;
    v  = Var_B1;
    v  = Var_B2;
    v  = Var_B3;
    v  = Var_C1;
    v  = Var_C2;
    v  = Var_C3;
```

To make the retrieval consistent, Var_P1 is taken to be the primary variable, belonging to the pointer file dat.p. Since the first two data files are already organized with respect to individual ID numbers, no pointers are needed for a match. But the third data file, dat.3, is organized with respect to household ID numbers, therefore we define a pointer between the household numbers in this data file (Var_C3) and the household numbers in the pointer file (Var_P2). This is exactly the setup defined in the archive description file exam.3.

The command rzoo cf=rr.3 results in the following output:

```
Var     Name          File        Offs   Len  Label
---  ----------  ------------  ------  ---  -----
  1  Var_P1      dat.p              0    1  1. variable of dat.p
  2  Var_A1      dat.1              0    2  1. variable of dat.1
  3  Var_A3      dat.1              9    2  3. variable of dat.1
  4  Var_B1      dat.2              0    1  1. variable of dat.2
  5  Var_B2      dat.2              1    2  2. variable of dat.2
  6  Var_B3      dat.2              4    4  3. variable of dat.2
  7  Var_C1      dat.3              0    2  1. variable of dat.2
  8  Var_C2      dat.3              3    4  2. variable of dat.2
  9  Var_C3      dat.3              7    2  3. variable of dat.2

Matching
   File: dat.1       Matching variable: Var_A1
   File: dat.2       Matching variable: Var_B2
   File: dat.3       Matching variable: Var_C1   Pointer: Var_P2
   File: dat.p       Matching variable: Var_P1

Primary file: dat.p (4)

1  1 90 1  1 1.01 90 9001  1
2  2 90 3  2 1.02 90 9001  1
3  3 91 5  3 1.03 91 9003  3
4  4 91           91 9003  3
5  5 91           91 9003  3
6  6 95 2  6 1.06 95 9008  8
7  7 95 4  7      95 9008  8
8  8 95 6  8 1.08 95 9008  8
9  9 97 8  9      97 9009  9

Number of records: 9
Memory used: 120314 bytes.
```

4.5 External Record Matching

The options of internal record matching, described above, are limited to two levels. On a first level the matching is done by directly using matching variables. A second level is introduced by pointer variables. If additional levels of matching are required,

RZOO provides for an option of external record matching. This option is also useful if two or more already existing data files have to be matched with respect to some of their variables.

The basic procedure in external record matching is the same as internal record matching via matching variables. The only difference is that with external record matching the data files are not taken from the archive but can be arbitrarily specified by the user. Suppose that two or more files shall be matched. Three pieces of information must then be given to the program:

1. For each of the files: its name.
2. For each of the files: an upper limit for the number of records to be read from the file. This information is necessary to prepare storage space for pointers to the records of the files.
3. For each of the files: the definition of a variable that shall be used as the matching variable.

The definition of matching variables can be done in two different ways, depending on the type of the file. If a file is written in a fixed format the matching variable is defined by the offset and length of the field the variable takes in the records of the file. It is the same way of definition as used in the variable description file explained in Section 2. If, instead, a file is written in a free format there are no fixed fields, but each variable can be identified by an ordinal number: the first variable, the second variable, and so on. The syntax for the specification of files is, respectively:

```
mf = n,off,len,fname    or    mf = n,c,fname
```

`fname` is the name of the file, `n` is the upper limit for the number of records to be read from the file. In the first case the matching variable is specified by the offset `off` and the field length `len`. In the second case the variable is specified by its ordinal number `c`.

As an example, the first three data files contained in the archive `exam.zoo` are matched. The following command file, called `m.1`, is used:

```
* m.1  Example command file for external record matching

    mf = 20,0,2,dat.1;   first (primary) file
    mf = 20,1,2,dat.2;   second file
    mf = 20,7,2,dat.3;   third file
    df = 1;              print to standard output
```

The resulting standard output is:

```
File: dat.1          Match:  Off   0  Len  2
File: dat.2          Match:  Off   1  Len  2
File: dat.3          Match:  Off   7  Len  2

File: dat.1          Records:    20 max     9 read     0 skipped
File: dat.2          Records:    20 max     8 read     0 skipped
File: dat.3          Records:    20 max     4 read     0 skipped

 1 -1.01 90 1 1 1.01 -1 90 9001 1
 3 -1.03 91 5 3 1.03 -5 91 9003 3
 8 -1.08 95 6 8 1.08    95 9008 8
 9 -1.09 97 8 9      -8 97 9009 9

Number of records: 4
Memory used: 514 bytes.
```

The primary, firstly defined file is taken as the reference. The output is always sorted with respect to the matching variable of this file. Then it is tried to find matching records from all other files. If this is successfull *for all files* a combined record is written to the output file, otherwise nothing is written. The combined record is simply a concatenation of the records from the files that take part in the matching. Separation is done according to the **sep**-command.

Note: With the current version of RZOO external record matching is only possible with files that have records defined by end-of-record characters.

5 Portability and Memory Requirements

As already mentioned in the Introduction, Rahul Dhesi's ZOO is a highly portable program. The program is running on many different platforms, and archive files can be transferred between different platforms and then processed by ZOO.[6] The same degree of portability should apply to RZOO. However, until now there is only limited experience.

Currently, RZOO has been tested on three different platforms. The program was developed on a UNIX SYSTEM V (3.2) workstation based on MOTOROLA's 88000 processor. It was then ported to a MACINTOSH SE/30 (SYSTEM 7.01) and to the MSDOS environment based on INTEL's 80386 processor. Compilation was done with several different compilers. It should be noted that DOS versions of the program are currently only available for 80386 machines. It is hoped that the program can be ported to other machines without difficulties.[7]

Memory requirements of the program depend on the type of jobs to be done. The program itself is relatively small, less than 120 kilobytes. However, this is without storage space needed to decompress files from the archive and for sorting and matching records. The program, written in the programming language C, makes dynamical memory requests from the operating system whenever it needs some storage space. And whenever some space is no longer needed it is given back to the system. If the high compression method is used, storage space needed for decompression of files is about 30 kilobytes *per file*. Additional memory is needed when records must be sorted. An estimate of these memory requirements is $(16 + 4\,m)\,n$ bytes where n is the maximum number of records involved and n is the number of files to be matched. If pointer variables are used one has to add $2\,n$ bytes.

This shows that memory requirements can become large if there are many data files and/or records to be retrieved simultaneously. Therefore, all currently available DOS versions of the program, based on 80386 machines, can take advantage of extended memory if it is available.

[6] Of course, archive files have to be transferred as binary files without any modification.

[7] Whenever new ports become available this will be recorded in the release notes for each new version of the program.

EUI WORKING PAPERS

EUI Working Papers are published and distributed by the
European University Institute, Florence

Copies can be obtained free of charge – depending on the availability of stocks – from:

The Publications Officer
European University Institute
Badia Fiesolana
I-50016 San Domenico di Fiesole (FI)
Italy

Please use order form overleaf

Publications of the European University Institute

To The Publications Officer
European University Institute
Badia Fiesolana
I-50016 San Domenico di Fiesole (FI)
Italy

From Name . . . . . .

Address . . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .

❒ Please send me a complete list of EUI Working Papers
❒ Please send me a complete list of EUI book publications
❒ Please send me the EUI brochure Academic Year 1992/93

Please send me the following EUI Working Paper(s):

No, Author . . . . . .
Title: . . . . . .
No, Author . . . . . .
Title: . . . . . .
No, Author . . . . . .
Title: . . . . . .
No, Author . . . . . .
Title: . . . . . .

Date . . . . . .

Signature . . . . . .

EUI Working Papers as from 1990

As from January 1990, the EUI Working Papers Series is divided into six sub-series, each series is numbered individually (i.e. EUI Working Paper HEC No. 90/1; ECO No. 90/1; LAW No. 90/1; SPS No. 90/1; EPU No. 90/1; ECS No. 90/1).

February 1992

Working Papers in Political and Social Sciences

SPS No. 90/1
Reiner GRUNDMANN/Christos MANTZIARIS
Habermas, Rawls, and the Paradox of Impartiality

SPS No. 90/2
Hans-Peter BLOSSFELD/Ursula JAENICHEN
Educational Expansion and Changes in Women's Entry into Marriage and Motherhood in the Federal Republic of Germany

SPS No. 90/3
Nico WILTERDINK
Where Nations Meet: National Identities in an International Organisation

SPS No. 90/4
Hans-Peter BLOSSFELD
Changes in Educational Opportunities in the Federal Republic of Germany. A Longitudinal Study of Cohorts Born Between 1916 and 1965

SPS No. 90/5
Antonio LA SPINA
Some Reflections on Cabinets and Policy-Making: Types of Policy, Features of Cabinets, and Their Consequences for Policy Outputs

SPS No. 90/6
Giandomenico MAJONE
Cross-National Sources of Regulatory Policy-Making in Europe and the United States

***

SPS No. 91/7
Hans-Peter BLOSSFELD
Is the German Dual System a Model for a Modern Vocational Training System?

SPS No. 91/8
Hans-Peter BLOSSFELD/ Gianna GIANNELLI/ Karl Ulrich MAYER
Expansion on the Tertiary Sector and Social Inequality. Is there a New Service Proletariat Emerging in the Federal Republic of Germany?

SPS No. 91/9
Giandomenico MAJONE
Public Policy Beyond the Headlines

SPS No. 91/10
Giandomenico MAJONE
Market Integration and Regulation: Europe after 1992

SPS No. 91/11
Jean BLONDEL
Ministers of Finance in Western Europe: A Special Career?

SPS No. 91/12
Jean BLONDEL
Governments and Supporting Parties: Definitions and Classifications

SPS No. 91/13
Jean BLONDEL
A Model for the Analysis of Government-Party Relationships

SPS No. 91/14
Jean BLONDEL
The Political Factors Accounting for the Relationship Between Governments and the Parties Which Support Them

***

SPS No. 92/15
Jerry A. JACOBS
Consumer, Social and Business Services Industries in the United States: Trends Between 1969 and 1987

SPS No. 92/16
Hans-Peter BLOSSFELD/
Yossi SHAVIT
Persisting Barriers: Changes in Educational Opportunities in Thirteen Countries

SPS No. 92/17
Hans-Peter BLOSSFELD/
Jan HOEM/Alessandra
DE ROSE/Götz ROHWER
Education, Modernization and Divorce. Differences in the Effect of Women's Educational Attainment in Sweden, the Federal Republic of Germany and Italy

SPS No. 92/18
Zina ASSIMAKOPOULOU/
Gøsta ESPING-ANDERSEN/
Kees VAN KERSBERGEN
Post-Industrial Class Structures: Classifications of Occupations and Industries (United States, Germany, Sweden and Canada)

SPS No. 92/19
Götz ROHWER
RZoo: Efficient Storage and Retrieval of Social Science Data